Saarland University
Faculty of Mathematics and Computer Science

Bachelor'sThesis

# Generating Infrastructural Code for Terms with Binders using MetaCoq and OCaml

**Author**
Dieter Schlau

**Advisor**
Prof. Dr. Fast Richtig

**Reviewers**
Prof. Dr. Fast Richtig
Prof. Dr. Leider Falsch

Submitted: 01$^{th}$ January 2017

### Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

### Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

### Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

### Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 01th January, 2017

# Abstract

Write an abstract.

Very nice thesis.

# Acknowledgements

# Contents

# Todo list

# Chapter 1

# Introduction

## 1.1 Overview

## 1.2 Related Work

## 1.3 Contributions

## 1.4 Coq

# Chapter 2

# Autosubst

## 2.1 De Bruijn Algebras

### 2.1.1 Substitution Equations

## 2.2 Autosubst Compiler

### 2.2.1 EHOAS

### 2.2.2 Dependency Analysis

### 2.2.3 Code Generation

## Chapter 3

# Autosubst OCaml

One part of the thesis is to port Autosubst to the OCaml language, which Coq is implemented in. We call this re-implementation **Autosubst OCaml**. To distinguish we call the existing implementation by Kathrin Stark **Autosubst 2**.
Analogous to Autosubst 2, Autosubst OCaml parses a user-supplied EHOAS language specification and generates Coq code necessary to prove substitution equations of that language in the form of Coq source files.

### 3.1 Reason to Exist

Autosubst 2 is written in Haskell and is distributed in source-form. This makes it necessary to maintain a Haskell environment if one wants to use the program. In order to represent the generated code, custom abstract syntax for Gallina terms, tactics and Coq vernacular commands had to be defined for Autosubst 2. Additionally, for each of these datatypes, pretty-printers to generate the concrete syntax had to be written from scratch.
With a re-implementation in OCaml users can re-use their existing OCaml environment for Coq to run the program and we can re-use the abstract syntax trees and pretty printers from the Coq implementation.

The ad-hoc nature of the abstract syntax and pretty-printers in Autosubst 2 is a disadvantage because for one it replicates functionality but also there are differences to the existing implementations from Coq, discussed in the following.
For example, the pretty-printers produce concrete syntax that uses more parentheses than necessary, has irregular spacing and is indented according to custom rules.

```
Definition upId_ty_ty  (sigma : ( fin ) -> ty ) (Eq : forall x, sigma x = (var_ty ) x)
  fun n => match n with
  | S fin_n => (ap) (ren_ty (shift)) (Eq fin_n)
  | 0  => eq_refl
  end.
Title: excerpt of code generated by \astwo{}
```

This makes it harder to read than code written by a human or printed by the standard pretty printers form the Coq implementation.
Also, we found one instance where non-parseable code is generated. This is possible due to a combination of an abstract syntax term which should not be possible to construct[1] and the pretty printer naively printing the malformed term. A Definition can not be recursive, so the "with" syntax for mutually recursion is not supported.

```
Definition subst_vl   (sigmavl : ( fin ) -> vl  ) := ...
with subst_tm   (sigmavl : ( fin ) -> vl  ) := ...

Title: Substitution operation if sort "vl" is declared before "tm" in System F
```

As for the abstract syntax, Autosubst 2 has an AST node[2] that contains function arguments which should be lifted in recursive calls. This bookkeeping behaviour is specific to syntax traversals and does not belong in an abstract syntax tree. While porting the code to use the AST from the Coq implementation, this became apparent as there was no analog to map this node to. When constructing a term we therefore use the contained function arguments directly.

cite traversal paper

Therefore, our plan for a re-implementation in OCaml is to reuse the abstract syntax and pretty-printers from the Coq implementation. This means we can only construct sensible abstract syntax terms, the generated code has a canonical look, and we can be more sure that it can be parsed again (albeit with the current design not that the code also typechecks).

## 3.2 How it Works

### 3.2.1 Parsing User Input

The user specifies their target language in our custom EHOAS for which we implemented a parser using the angstrom parser combinator library.

```
ident          :: [a-zA-Z][a-zA-Z0-9'_]*
spec           :: sortDecls functorDecls constructorDecls
sortDecl       :: ident ':' 'Type'
                | ident '(' ident ')' ':' Type
...
Title: EHOAS syntax
```

complete EHOAS specification

[1] If the first sort in a component of size $\geqslant 2$ is not recursive, the substitution operation is falsely generated as a Definition instead of a Fixpoint

[2] called TermSubst

We support most of the features of EHOAS from Autosubst 2, like first-order binders, functors and parameters.. The exception is we do not support modular syntax as it is a complex extension that is out of scope for this work.
We implemented some improvements over the Autosubst 2 parser, namely:

note that it's not really HOAS because no higher-order bidners? How did Kathrin do it?

- Support the same identifiers that Coq allows, i.e. containing unicode, by reusing functions from the Coq parser.
- Allow a user to specify custom name of variable constructor when declaring a sort. The name was auto-generated in Autosubst 2
- A sanity check that identifiers are not declared twice.

The dependency analyses of the parsed specification is implemented with the `ocamlgraph` graph library and works analogously to Autosubst 2 with one extension. We already compute which sorts have a renaming during this stage while Autosubst 2 has a function for that which it calls during code generation. The reasons are that it is more efficient, the Haskell code calls the function many times but OCaml does not do the same memoization, and the function uses non-structural recursion. Avoiding this makes it easier to port the code to MetaCoq later.

### 3.2.2 Usage of Coq as a Library

To use the exposed facilities of the Coq implementation we need to link the implementation's modules from our own OCaml code. This then allows us to use the Coq parser, several abstract syntax trees, and pretty printers for terms.

> One roadblock was that when linking certain modules of the Coq implementation one needs to add the `-linkall` flag during compilation which causes all modules of the Coq library to be linked in the resulting executable. This is because these modules depend on other modules to set up state. If `-linkall` is omitted, the state is not initialized correctly and the program will crash at runtime.

footnote linking emilio's explanation why that is the case

#### Gallina Term AST

The Coq implementation exposes multiple ASTs for Gallina terms, which we use to construct the types and proof terms of the generated lemmas, and one AST for vernacular commands, which we use to construct `Inductives`, `Definitions` and `Lemmas` that use the the aforementioned types and proof terms.
The three term ASTs are

- `constr_expr`
- `glob_constr`
- `econstr`

When Coq runs, a term that the user intputs is first parsed into the `constr_expr` form and then translated down into the other forms. So `constr_expr` is the AST most closely related to concrete syntax, `econstr` is the AST that the kernel uses for typechecking, and `glob_constr` is an intermediate form.

We opted to use the `constr_expr` AST to construct Gallina terms because it is simpler to construct and easier to use with other parts of the code than the other two ASTs due to the following reasons.

- Other functions that we want to use, like the pretty printers and constructors for vernacular commands, use `constr_expr`.
- It has variadic lambdas and applications whereas the other two have the curried forms, meaning the application node applies the function only to a single argument.
- It uses named variables instead of De Bruijn indices for locally bound variables.
- It uses named variables instead of references into the environment for globally bound variables.

We also define smart constructors to take care of boilerplate in the `constr_expr` AST. These constitute the DSL in which we construct all Gallina terms.

> To be more precise about the last point, a globally defined variable (e.g. defined by `Definition` or `Parameter` etc.) consists of a kernel name and a universe instance. The kernel name is the fully qualified name to the constant, e.g. `"Coq.Arith.PeanoNat.Nat.add"` and the universe instance specifies the universe levels of the constant's arguments.
> So to construct a term for `add x y` we would have to know the fully qualified name of `add`, set out fitting universe levels (if any) and calculate the De Bruijn indices for `x` and `y`, assuming they are locally bound.

Especially the last two points show that it is simpler to construct terms of `constr_expr` than terms of the other ASTs. For one, because while De Bruijn indices work well in proofs, programming with them is complicated and error prone.

cite?

And second, the other ASTs are **globalized**, meaning all globally bound variables (e.g. `nat` or one of our `Lemmas`) are represented by references into the environment. To fill that environment we would have to start a Coq instance and parse the prelude, and also put any of our defined constants we need to reference in other terms in it.

In a discussion on Zulip we were advised against this because starting a Coq instance and handling the environment in our program would also be error prone.

link Emilio's post

The only advantage of using one of the globalized ASTs – albeit a major one – is that we could use the kernel's typechecker to make sure our constructed terms are valid. This prevents a situation where we construct uncompilable source code. However, we decided to go with the `constr_expr` AST to keep or program simple and from experience with Autosubst 2 we know that the program does not construct untypable terms.

call it future work?

**Vernacular Command AST**

Vernacular commands are represented by `vernac_expr` terms which often contain `constr_expr` terms, e.g. a `Definition` has a type and a body which are `constr_exprs`. In our program we need to construct and print the commands

`Inductive`, `Definition`, `Lemma`, `Proof`, `Qed`, `Notation`, `Typeclass`, `Instance`, `Ltac`.

Most of these are straightforward to construct and use (except `Proof` and `Ltac`, see below) but we define smart constructors to take care of boilerplate.
We only found one instance where the pretty-printer of Coq was wrong and printed superfluous commas between arguments of the `Existing Instances` command. This has been reported to the Coq bugtracker.

link & update when I do pull request and in case I find more

> One irregularity when constructing a `Proof` command is that there is the `VernacExactProof` node to give a proof term directly with the command. It is pretty-printed as
>
> ```
> Proof (foo).
> ```
>
> But proof-general does not work with this syntax and there is discussion to deprecate it entirely so we decided to hook a custom pretty-printer which instead prints
>
> ```
> Proof.
>   exact (foo).
> ```
>
> Another solution would be to always use `Definitions` which don't need the `Proof` command at all.

link proof general error

**Tactic AST**

Tactics also have multiple ASTs and we opted again to use the one closest to the concrete syntax, `raw_tactic_expr`. Combinators like `try`, `repeat`, `progress` and basic tactics like `rewrite`, `cbn`, `unfold` are part of this AST. We again define smart constructors to take care of boilerplate and act as a DSL which allows us to construct tactics in a natural way.[3]

mention vok compilation here?

```
let rewrites = List.map (fun t -> try_ (setoid_rewrite_ t)) substify_lemmas in
```

[3] then_ tac1 tac2 is tactic sequencing, i.e. "tac1; tac2"

```
let tac = then_ (calltac_ "auto_unfold" :: rewrites) in
TacticLtac ("substify", tac)
```

Other tactics like `setoid_rewrite` were added using Coq's syntax extension mechanism so they are not a native part of the AST and we cannot construct them directly. We can work around this fact because `vernac_expr` allows us to reference a tactic by a string (similar to how you can refer to any constant with in `constr_expr`). Semantically this is meant for tactics defined by an `Ltac` command, but it works for `setoid_rewrite` because the concrete syntax is the same.

> The canonical way to construct a term representing a `setoid_rewrite` is to define the syntax extension, put it in the environment and reference it with the `TacML` AST node. But to again avoid manipulation of the environment we instead `TacCall` node, used for calling user-defined tactics, which results in the same concrete syntax `"setoid_rewrite foo"` albeit the abstract syntax term has a different intended meaning.
> The only problem is that we cannot give a rewrite orientation this way because ← is not a valid identifier in an argument position of `TacCall`. We define a left-rewrite version of `setoid_rewrite` and reference this instead.
> `Ltac setoid_rewrite_left t := setoid_rewrite <- t.`
> Due to the combinatory explosion if there are many flags this is not suitable in general, but in our case this was the only special case.

This AST only allows us to construct the tactic terms. Commands like `Ltac` to bind a tactic term to a name are also defined using Coq's syntax extension mechanism so there is no easy way to construct and print it. Therefore, we define our own trivial printer for this which only needs to print `Ltac some_name :=`, the rest is delegated to the Coq's pretty-printer for tactic terms.

### 3.2.3 Generation of Abstract Syntax

## Chapter 4

# Autosubst MetaCoq

As a continuation of our efforts to re-implement Autosubst 2 in OCaml we also planned to re-implement it in the MetaCoq framework. This implementation is then called **Autosubst MetaCoq**.

cite

### 4.1 Reason to Exist

While a user interacts with Autosubst OCaml on the command line and receives source code files they can use in their development similar to Autosubst 2, an implementation in MetaCoq allows a seamless interaction from within a running Coq instance.
Apart from the usability standpoint, it part of a line of work exploring the metaprogramming capabilities of the MetaCoq framework.

cite line of work

### 4.2 How it Works

MetaCoq allows a user to **unquote** an abstract syntax term, which is turning a reified term into a Gallina value it describes [1] and interact with the Coq environment by registering new inductive types and definitions. Both is enabled by side-effects produced by evaluating a `TemplateMonad` expression.

Therefore, we can implement Autosubst by computing abstract syntax terms of types, proof terms and inductive types, analogously to Autosubst 2 and Autosubst OCaml, using the provided AST by MetaCoq [2] and constructing a `TemplateMonad` expression that unquotes those terms and puts them into the environment.
The architecture of the program is then mostly the same to Autosubst OCaml but we replace the pretty-printers that give us a string representation of an abstract

[1] This is done using the `tmUnquoteTyped : ∀ A, term -> TemplateMonad A` constructor which takes a target type along with an abstract syntax term and tries to unquote it into the Gallina value it describes
e.g. turning the Gallina value `"tApp eq_ [nat_; one_; one_ : term"` into the Gallina value `"@eq_refl nat 1 : 1 = 1"`

[2] called `term` which is rather nondescript, so we call it the MetaCoq AST

syntax term with MetaCoq functions that unquote such a term.

### 4.2.1 Parsing User Input

Because the user will interact with Autosubst MetaCoq in a running Coq instance we cannot receive the language specification in a text file like in previous implementations. There are multiple possible ways how the user could transmit the language specification to our program.

#### Inductive Types

One possibility is that a user implements the inductive types for their language directly and we analyze them using MetaCoq.

```
Definition bind {X Y:Type} (a: X) (b: Y) := b.

Inductive tm : Type :=          Inductive tm (n: nat) : Type :=
| app : tm -> tm -> tm         | app : tm n -> tm n -> tm n
| lam : (bind tm tm) -> tm.    | lam : (bind (tm 0) (tm (S n))) -> tm n.
```

But there are multiple problems with this approach.

- Do we reuse the inductive type the user writes?
  If yes, the user would have to additionally specify a variable constructor and scope parameters if necessary (like example on the right).
  If no, there would be two similar copies of the inductive types.
- Coq only supports strictly positive recursive occurrences in an inductive type definition[3] so specifying the bound sort is awkward[4].
- Coq supports more syntax for inductive types like `Cumulative`, `Polymorphic`, and indices which we have to take care to not accept.

#### Custom Entry Notations

cite the paper by herbelin

Another possibility is embedding a DSL for our EHOAS language in Coq using custom entry notations. With custom entry notations one can replace the Coq parser inside of designated symbols (here "`{{`" and "`}}`") so that one can define their own interpretation for tokens.
In our case, the notations desugar to some inductive types modelling EHOAS, analogous to those after the parsing step in Autosubst OCaml. To model EHOAS we

[3] in the example above, we cannot have a constructor of this type `bad : (tm -> tm) -> tm` because the first `tm` is to the left of an arrow in the type of the argument, which is a "negative" position

[4] bind ignores the first argument because when use the lam constructor we want to call it with only one argument, `tm (S n)`. But we still need the first argument in the abstract representation of the constructor type to know the bound sort.

want to redefine parsing for ':' and '->' to not be a typecast or function arrow, respectively, and allow any identifier as construct and sort names[5].

```
Definition stlc_ctors := {{ app : tm -> tm -> tm;
                            lam : (bind tm in tm) -> tm }}.
```

```
Title: How to specify constructors for the STLC
```

This allows a natural input syntax that looks almost the same as in the other implementations.

Dependency analysis is based on the same graph analysis as with Autosubst 2. Except we also precompute the sorts that have renamings like in Autosubst OCaml because the that computes this in Autosubst 2 does not use structural recursion so it would be harder to port. For the graph analysis we had to write our own graph representation with a couple of related algorithms as the existing implementations in the Coq ecosystem were not suitable for computation.

list which ones: mathcomp & the one form coq-community iirc

### 4.2.2 Programming in MetaCoq

Because Gallina is a functional language whose syntax is inspired by ML we can use almost the same architecture as with Autosubst OCaml. Some concessions had to be made because working with the MetaCoq AST works differently than the `constr_expr` AST we use in Autosubst OCaml but behaves more like the `constr_expr` AST. This means

- References are not strings but pointers into the environment. Constructing these references is more involved than just using a string but we can use MetaCoq functionality to turn a string into a reference.

  At the moment this is done by using the `tmLocate` functionality of the `TemplateMonad` which turns a name into a reference. This necessitates an architecture of our program where we alternate between the `TemplateMonad` and our `GenM` Monad, where code generation happens.

  diagram that shows the relationship of the TemplateMonad and our GenM monad

- Locally bound variables are not strings but De Bruijn indices. This has the downside that it is harder to program in. How we deal with this is explained below.
- With the MetaCoq AST it's not possible to write a function with implicit arguments. We also explain below how to work around this.

[5] The fact that we can write the identifier 'app' instead of the string '"app"' in the notation is a hack described in [CITE]

### Implicit Arguments

Because the MetaCoq AST does not contain information about which arguments of a function are implicit, we cannot generate functions with implicit arguments. If one writes a function application in Coq and wants to treat certain arguments as implicit, one can use underscores in their place like `(f _ a _ b)`.
In the MetaCoq AST we can use an analogous node, `tHole`, and then construct function applications that explicitly pass these holes in place of arguments that should be implicit like `(tApp f [tHole; a; tHole; b])`.

> Underscores are concrete syntax for existential variables (**evars**), which are typed placeholders for some other term. The MetaCoq AST has a node for evars, called **tHole**, so when we generate a function application we can put `tHole`s in place of implicit arguments which turn into evars after unquoting and are then inferred by Coq.
> This is the same way that implicit arguments work in Coq except you don't have to explicitly pass a placeholder in Coq.
> Evars in a term necessitate the use of `tmUnquoteTyped` to unquote MetaCoq AST terms. There is also the `tmUnquote` command which infers the return type but it fails if the given term contains evars.

### De Bruijn Indices

The biggest difference to `constr_expr` is that the MetaCoq AST uses De Buijn indices for local variables. While De Bruijn indices work well in proofs, programming with them is complicated and error prone. When generating expressions with De Bruijn indices, one way of dealing with them is to use counters to keep track of under how many binders a given term is. This results in code like this which is both hard to write and understand.

how to quote Marcel's code? is from source/destruct_lemma.v::559

```
let fpos := 1+allIndCount in
let Hstart := fpos in
let Hpos := Hstart+(ctorCount -i) in
let ppos := 1+Hstart+ctorCount in
let paramOffset := 1+ppos in
let recPos := paramOffset + trueParamCount in
...
```

Another way is to use environments that map a variable name to its index. Then you can use the name when constructing the term and must take care to update the environment to be in sync with the term that you want to construct at a given time. Keeping the environment in sync turns out to be a major disadvantage, like in the following example where you have to thread the environment through the fold. It would be an option to make all the code monadic, though, to take care of this.

```
code example
```

Our solution was to implement a custom AST that is identical to the MetaCoq AST except that it has named variables. This allows us to build a closed expression and then translate all the named variables to De Bruijn indices in one go which makes the code generation functions much easier and in line with our Autosubst OCaml implementation.

comparison in figure

```
Inductive nterm : Type :=
| nRef : string -> nterm (* turns into tRel, tConst, tInd, tConstruct from the term typ
| nHole : nterm
| nTerm : term -> nterm (* embed MetaCoq AST terms to speed up translation *)
| nProd : string -> nterm -> nterm -> nterm
| nArr : nterm -> nterm -> nterm
| nLambda : string -> nterm -> nterm -> nterm
| nApp : nterm -> list nterm -> nterm
| nFix : mfixpoint nterm -> nat -> nterm
| nCase : string -> nat -> nterm -> nterm -> list (nat * nterm) -> nterm.

Inductive term : Type :=
| tRel : nat -> term
| tVar : ident -> term
| tEvar : nat -> list term -> term
| tSort : Universe.t -> term
| tProd : aname -> term -> term -> term
| tLambda : aname -> term -> term -> term
| tApp : term -> list term -> term
| tConst : kername -> Instance.t -> term
| tInd : inductive -> Instance.t -> term
| tConstruct : inductive -> nat -> Instance.t -> term
| tCase : (inductive Œ nat) Œ relevance ->
          term -> term -> list (nat Œ term) -> term
| tFix : mfixpoint term -> nat -> term
| ...

Title: Our AST "nterm" in comparison to the MetaCoq AST (abridged)
```

### 4.2.3 Generation of Abstract Syntax

# Appendix A

# Appendix

List of types and functions we use from the Coq implementation

- `constr_expr`

# Bibliography